Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Wissenschaftliches Rechnen

Christoph Sprenger

User's Guide to SCIDDLE Version 3.0

December 1993

208

Departement Informatik
Institut für Wissenschaftliches Rechnen
Prof. Dr. W. Gander

Christoph Sprenger
Laboratoire de téléinformatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne
e-mail: sprenger@di.epfl.ch; sciddle@inf.ethz.ch

This report is also available via anonymous `ftp` from `ftp.inf.ethz.ch`
as `doc/tech-reports/1993/208.ps`.

© 1993 Departement Informatik, ETH Zürich

Contents

1 Introduction **1**

2 Compiling and Installing SCIDDLE **4**
- 2.1 Obtaining the SCIDDLE Package 4
- 2.2 System Requirements 4
- 2.3 Compiling SCIDDLE 5
- 2.4 Setting Up the Super-Servers 5

3 Getting Started: A Simple Example **8**

4 Language and Stub Compiler **12**
- 4.1 Constant Declarations 12
- 4.2 Type Definitions 13
- 4.3 Context Variable Declarations 14
- 4.4 Procedure Declarations 14
- 4.5 Array Handling 15
- 4.6 Running the Stub Compiler 17
- 4.7 Calling Remote Procedures 18

5 Runtime Library User Interface **21**
- 5.1 Service and Server Management 21
- 5.2 Call Management 22
- 5.3 Miscellaneous Functions 24
- 5.4 Error Codes and Messages 25

6 The Monitor **26**
- 6.1 Display Mode 26
- 6.2 Command Mode 26

7 Writing and Building your Applications **27**

8 Running your Applications **28**
- 8.1 Checking the Super-Server Setup 28
- 8.2 The Services File `.sc_services` 28
- 8.3 Running the Monitor 29
- 8.4 Starting the application 29

9 Example: Parallel Distributed Matrix Multiplication **30**

A Interface Definition Language Syntax **34**

B Stub Compiler Error Messages **37**

C Stub Reference **38**

D User Library Reference **49**

E Runtime System Error Codes and Messages **87**

F File Formats **88**

1 Introduction

This User's Guide to SCIDDLE Version 3.0 explains how to obtain a copy of the SCIDDLE software, how to compile and install it and the steps to take when writing and running a SCIDDLE application.

SCIDDLE is a remote procedure call (RPC) package that enables easy programming of distributed applications on a network of loosely coupled processing nodes. The nodes participating in a distributed computation might well be a heterogeneous collection of machines, ranging from workstations over shared memory multiprocessors to vector supercomputers. These machines might be interconnected by different networks, such as Ethernets, Internet and fiber optic networks.

All interprocess communication in SCIDDLE is based on remote procedure calls. Remote procedures are similar to ordinary (local) procedures, except that the calling and the called program parts are executing in two different address spaces (and so possibly on two different computers). The procedure arguments and results have to be transmitted between the two address spaces over some communication medium (a local area network, for example). It is the aim of every RPC system to make this fact as transparent as possible to the user.

Since procedures are a well-understood paradigm to structure sequential programs, remote procedures provide a convenient way to write distributed applications, too, and they allow the programmer to think in abstractions known from sequential programming.

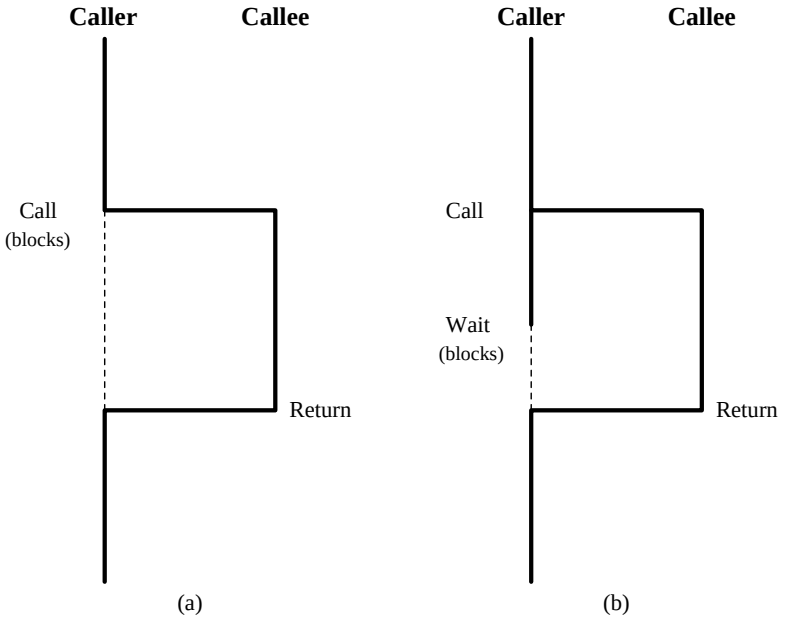

Figure 1: RPC control flow: (a) synchronous and (b) asynchronous

SCIDDLE does not only support distribution, but also parallelism. In fact, it was particularly designed to support concurrent application programming with RPCs. Therefore, besides ordinary synchronous RPC, it provides *asynchronous* or *non-*

blocking RPC. Asynchronous RPCs do not block the calling process while the remote procedure is executed. Thus, multiple remote calls might be issued before results are collected. With synchronous RPCs, the caller (client) transfers control to the callee (server) and is blocked until results arrive. Figure 1 depicts the control flow of the two types of RPC schematically. Synchronous RPC follows a strict sequential control flow, whereas parallelism is inherent in asynchronous RPC. [Of course, using synchronous calls, parallelism could be accomplished by creating a separate thread of control for each call on the caller side. But, first, this complicates the caller program, and second, to our knowledge there is no thread package available under UNIX which is portable and does not block the whole process when doing I/O. The latter may still enforce an undesirable serialization of the calls.]

Programming with RPCs implies a client-server model of computation. SCIDDLE allows servers to be clients of other servers. Communication between servers is not possible in this model. Consequently, the process dependency graph has the shape of a tree. A client may claim several different *services*. A service is declared by a *remote interface definition*. A remote interface definition consists of a set of constant and procedure declarations and completely specifies the application-level communication protocol between client and server.

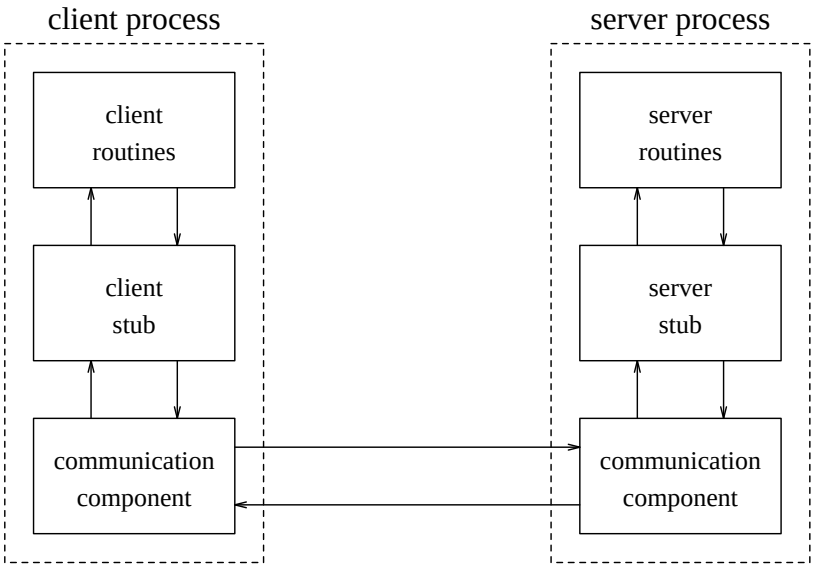

Figure 2: Remote Procedure Call

The SCIDDLE *stub compiler* translates remote interface definitions into two *stub modules* (or briefly *stubs*) - a *client stub* and a *server stub*. They contain the code needed to transmit the procedure parameters from the client over the network to the server and vice versa. In fact, the stubs transform local procedure calls into request and reply messages which are transmitted over the network by a communication component. Figure 2 shows the components involved in a RPC and their interrelationship. The programmer is not concerned with writing network-related code. He only supplies the interface definition(s) and the client and server modules.

In general, only minor modifications are necessary to turn a sequential application into a distributed one. The interface definition language and the stub compiler are described in section 4.

The SCIDDLE *user library* contains routines for service initialization, server process startup and termination, management of on-going asynchronous calls, memory management and some other functions. It is described in section 5.

Server processes are started through a user library call from the client process in cooperation with a *super-server* running on each server machine. Servers are private to the client that started it up. This means that only that client may issue remote calls on the server. Remote call requests are queued at the server and processed iteratively. A concurrent application that needs many (e.g. compute) servers usually creates a *server pool* at initialization time. You will learn about how to install super-servers in section 2.4.

Servers are short-lived which means that their lifetime is bound by the client's lifetime. Normally, the client terminates a server when it does no longer need its service. However, for if the client forgets to do so, terminates abnormally or its machine has crashed, servers periodically check for the client's liveliness. When a server detects that the client has disappeared, it terminates itself. Such a server is termed an *orphan*. The *orphan detection and killing mechanism* saves system resources and ensures a proper cleanup of the whole system in the presence of failures.

The final SCIDDLE component is the *monitor*. It collects all terminal output from the servers of a running application and displays it on a single terminal. It is also used to enter passwords, if some super-servers request one. Moreover, the monitor's command mode provides primitive operations for super-server management.

A special feature was introduced in order to support the programming of concurrent numerical applications. These applications frequently require parts of matrices and vectors to be distributed among a group of processors. SCIDDLE supports this task by providing *views* which can be attached to array-typed parameters and serve to select the actual subarray that will be transmitted in a remote call.

2 Compiling and Installing SCIDDLE

The SCIDDLE software is composed of four parts: the stub compiler (**stubgen**), the user library (**libsc.a**), the Fortran-to-C interface conversion library (**libf2c.a**), the super-server (**scidd**) and the monitor (**scmon**). The following sections describe how to acquire a copy of the SCIDDLE package and how to compile and install it on your system.

2.1 Obtaining the SCIDDLE Package

SCIDDLE can be obtained via anonymous ftp from **maggia.ethz.ch** (internet address 129.132.17.1). Copy the file **sciddle3.0.tar.Z** from the directory `/pub` to your home directory. This is a compressed tape archive and can be unpacked by typing the following command[1]:

```
% zcat sciddle3.0.tar.Z | tar -xof -
```

2.2 System Requirements

SCIDDLE runs under the UNIX operating system. It needs the following non-standard (non-System V) Berkeley and SunOS operating system features:

- Berkeley sockets with the TCP/UDP/IP protocol suite
- Sun XDR (eXternal Data Representation, part of Sun RPC)

Today, most UNIX flavors are extended by these quasi-standard networking capabilities. SCIDDLE currently runs on the following machines and UNIX flavors:

Tag	**Architecture**	**UNIX flavor**
SUN3	Sun 3	SunOS
SUN4	Sun SPARCstation	SunOS
RS6K	IBM RS/6000	AIX
SGI	Silicon Graphics	IRIX
CRAY	Cray Y-MP	UNICOS
SX3	NEC SX-3	SUPER-UX

Table 1: Current implementations

Ports to other machines and UNIX flavors usually require only minor modifications to the source code, which are mostly related to some header files and accessing the password file.

[1] If your system does not understand the o-option for `tar`, leave it out

2.3 Compiling SCIDDLE

The SCIDDLE package is written in C, the stub compiler uses `lex` and `yacc` for its front-end. In order to compile the SCIDDLE system, type

```
% cd sciddle3.0/src
% MkSciddle3 <ARCH>
```

where `<ARCH>` is the architecture tag of your machine as shown in Table 1.

If your architecture is not listed in that table, you have to create a new architecture directory and make a copy of the generic makefile. Do this by typing the following commands:

```
% cd sciddle3.0/src/<sciddle-component>
% mkdir <ARCH>
% cd <ARCH>
% cp ../Makefile.proto Makefile
```

The architecture directories help you to manage different object codes, if the same account gives you access to several different machine architectures sharing a common file system (such as Sun NFS). Repeat this sequence of commands for each SCIDDLE component `<sciddle-component>` (i.e. `stubgen`, `runtime`, `libf2c` and `daemon`). The only thing you definitely need to change in the `Makefile`s is the architecture tag. Furthermore, they may have to be customized by setting some architecture-specific compiler/loader flags. As already mentioned, some modifications may be required in order for SCIDDLE to work on other machines.

The relevant executables and libraries are accessible via the following paths:

```
super-server:   sciddle3.0/src/daemon/<ARCH>/scidd
monitor:        sciddle3.0/src/monitor/<ARCH>/scmon
user library:   sciddle3.0/src/runtime/<ARCH>/libsc.a
f2c library:    sciddle3.0/src/libf2c/<ARCH>/libf2c.a
```

You may want to include the former two in your path environment or make links to them from a local `bin` directory already included in the path.

2.4 Setting Up the Super-Servers

Server processes are started by the SCIDDLE super-server. You have to install a super-server on each machine where you want to run server processes. You only need to do this once, because once you have started a super-server it will remain there even if you log out. The super-server program is called `scidd`. It understands the following command line options

```
-p <port#>    listen at specified port instead of default port
-l <logfile>  log to specified file instead of default file
-s            run in secure mode (always ask for password)
```

Normally, the super-server writes to a logfile named `scidd.log.<pid>` in the directory `/tmp`, where `pid` is its process ID. You may specify a different log file with the `-l` option. If the super-server fails to process some request (such as starting up a server), the log file can sometimes provide additional information about the reason of the failure.

From the security point of view, the super-server normally behaves just like `rsh` or `rlogin`, i.e. it checks for an entry for the client host in the `/etc/hosts.equiv` and `.rhosts` files. If there is one, the requested operation (f.e. startup) is performed without asking for a password. If there is no such entry the user is prompted for a password via the monitor.

However, on some machines (like supercomputers) this authentication scheme is too weak. There, you may want to run the super-server with the secure option `-s`. In this mode the super-server ignores the `/etc/hosts.equiv` and `.rhosts` files. It always asks the user for a password before it starts up a server.

The super-server can run in two different modes:

- single-user mode
- multi-user mode

If you start a super-server yourself (i.e. as a user), it automatically runs in single-user mode and serves only your requests and rejects any requests from a different user. If you want to share a super-server with several users, you may ask your system administrator to run the super-server as a root process. In this case, it is in multi-user mode and accepts requests from any user and sets the correct user and group IDs when starting a server for you.

There are two ways to start a super-server on a remote machine: you can use the SCIDDLE monitor or you can do it by hand.

Using the Monitor for Setup

Installing super-servers with the monitor works only, if you can execute commands with `rsh(1)` (`remsh(1)` on some machines)[2] and if the path to the super-server is included in the `PATH` environment variable on the remote machine. If this does not work on some machines, start the super-server "by hand". This is described in the following subsection.

Start the monitor by typing `scmon` and put it into command mode by pressing `<Return>`. Now, you should see a prompt (%) and you can start a super-server on a remote machine with the command `start <machine>`. If you want the super-server to listen at a particular port, specify also the port number on the command line. There are two special port numbers: '`#`' means the multi-user mode default port and '`*`' means the single-user mode default port. Additionally, you may set one of the options listed above. The `-p` option is equivalent to specifying the bare port number. After each command, you have to type an additional `<Return>` to put the monitor back into command mode (see also section 6).

[2] check the `.rhosts` file for an appropriate entry

"Manual" Setup

In order to start a super-server "by hand", log in on the remote machine, and type

```
% cd sciddle3.0/src/daemon/<ARCH>
% scidd [<options>]
```

where <ARCH> is the architecture of the machine and <options> are the options listed above.

The Host Configuration File `.sc_hosts`

The super-server prints out a port number as it starts up. This port number is later used by the runtime system to contact the super-server, so we have to remember it. If you lost the port number of some super-server, you always can find it at the beginning of its log file. Port numbers as well as login names are entered in the *host configuration file* `.sc_hosts`. This file has entries of the form

```
<host-name>   <port-number>   <login>
```

and must be located in your home directory. There are two special port numbers: '#' means the multi-user mode default port and '*' means the single-user mode default port. Empty lines and lines starting with the hash mark (#) are ignored. You need a copy of this file on all client machines.

3 Getting Started: A Simple Example

Now, as you have installed SCIDDLE and set up the super-servers, we want to implement and run a simple example program sending a string to a server which prints it out. The output will be displayed on the SCIDDLE monitor. We start with the remote interface definition. Here it is:

```
INTERFACE Hello;
(* Hello.sc - interface definition for Hello example *)

PROCEDURE SayHello(IN str: STRING): SYNC;

END
```

The server program is very simple. It does nothing else but printing the string it received from the client:

```
/* HelloServer.c - server program for Hello example */

#include <stdio.h>
#include "Hello_srv.h"

void SayHello(str)
char *str;
{
    printf("client says: %s\n", str);
}

main()
{
    setup_Hello();
    run_Hello();
}
```

Note that we also have to implement the server main program, which just calls the two routines `setup_Hello()` and `run_Hello()` exported by the server stub. Note the naming convention for these routines. They are composed of the prefixes `setup_` and `run_` plus the name of the interface definition (`Hello`) appearing after the keyword `INTERFACE`. The server program needs to include the server header file `Hello_srv.h` generated by the stub compiler.

Now comes the client program. This is slightly more complicated as we have to do some initialization and start up the server. Therefore, we need to call a couple of routines from the SCIDDLE user library.

```
/* HelloClient - client program for Hello example */

#include "sciddle.h"
```

```
#include "Hello_clt.h"

#define  SERVICENAME    "HELLO"
#define  SERVERMACHINE  "ru9.inf.ethz.ch"
#define  HELLOSTRING    "Hi, I am Sid Barrett and who are you ?"

main(argc, argv)
int argc;
char **argv;
{
    int srvid;

    /*
     * initialize master and attach HELLO service
     */
    sc_initmaster(&argc, argv);
    sc_svcattach(SERVICENAME, init_Hello);

    /*
     * start up a HELLO server on the indicated machine
     */
    srvid = sc_srvstart(SERVERMACHINE, SERVICENAME, 0);

    /*
     * call the remote procedure
     */
    call_SayHello(HELLOSTRING, srvid);

    /*
     * terminate server and detach service from client
     */
    sc_srvterm(srvid);
    sc_svcdetach(SERVICENAME);
}
```

Now, let us go through this program step-by-step. The client program needs to include `sciddle.h` for the the user library interface definitions and the client header `Hello_clt.h` produced by the stub compiler. The first call to `sc_initmaster()` initializes the master (the top-level client). It takes the command line arguments as parameters and filters out those interpreted by SCIDDLE. For instance, you can specify the machine and port where the SCIDDLE monitor is listening. We will learn about the monitor soon when running this program. The next statement `sc_svcattach()` registers the client stub with the runtime support. A string identifier is passed and assigned to the service. It is be used later to start up servers (instances of the service), . The second parameter is the client stub initialization routine `init_Hello()`. The naming convention for this routine is analogous to the one used for the setup and run routines in the server stub (see above).

Now, the runtime system is initialized and we are ready to start a `HELLO` server. The function `sc_srvstart()` initiates a server process on the indicated machine. The second argument is the name of the service and the third is a nice value allowing to run servers with lowered priority. On success, this call returns a server identifier.

At this point, the server is started and ready to accept remote call requests from its client. We call the remote procedure `SayHello` by an invocation of the stub routine `call_SayHello()` passing a string and the server identifier as arguments. Note that there are only two minor differences from a local call. First, the remote procedure name is prefixed by `call_` and second there is an additional argument at the end of the argument list, the identifier of the server to be invoked. Apart from that, the call looks like an ordinary local call, doesn't it? Since this is a synchronous procedure, the call returns as soon as the server has processed it.

Next and last, we terminate the server with `sc_srvterm()` and detach the service from the client with `sc_svcdetach()`.

Compiling and running the program. The first thing to do is to run the SCIDDLE stub generator on the interface definition file `Hello.sc`. This should look like the following

```
% stubgen Hello
Sciddle Stub Generator V3.0a
- Hello.sc
+ Hello.h
+ Hello_clt.h
+ Hello_clt.c
+ Hello_srv.h
+ Hello_srv.c
No errors
```

Next, compile the client program `HelloClient.c` and stub `Hello_clt.c` and link them together with the `Sciddle` user library `libsc.a`. Then, compile the server program `HelloServer.c` and the server stub `Hello_srv.c` and also link them with the user library.

Before we can run the distributed Hello program, we need to enter the `HELLO` service in the service file `.sc_services` in the home directory of the server machine. The super-server accesses this file to get the name of the executable for the service on its machine. Empty lines and lines starting with a hash mark (`#`) are ignored. The corresponding lines in this file could look like

```
# Hello service
HELLO     sciddle3.0/Hello/%/HelloClient
```

The service name is the same we used in `sc_svcattach()` to attach the the client stub. The relative executable path starts from the user's home directory. The percent sign (`%`) in the path is replaced with the architecture tag of the server machine.

Now, start the SCIDDLE monitor `scmon` on the same machine where you are going to run the client. The monitor will print a message like

```
Sciddle Monitor V1.0
listening on default port
```

We are ready to start the client program. The output of the server will appear in the monitor window, prefixed with the name of the server machine:

```
Sciddle RTS V3.0 <pid=1423>.
ru9.inf.ethz.ch: Hi, I am Sid Barrett and who are you ?
```

If you start the monitor on a different machine (not the client machine), you need to tell the client, where it can find the monitor. This is also true, if the monitor can not use the default port. In this case, start the client with the option `-m <mon-machine> <mon-port>`.

That's it. In the following sections we describe in more detail how to use SCIDDLE. We begin with the interface definition language and the stub generator.

4 Language and Stub Compiler

The SCIDDLE *remote interface definition language* is a declarative language that serves to specify the set of procedures exported by a particular service. It looks much like a definition module in Modula-2. The complete syntax of the language is given in appendix A. An interface definition is composed of four different sections. There are constant declarations, type definitions, context variable declarations and procedure declarations. A skeleton interface definition looks like

```
INTERFACE <interface-name>

CONST
    <const-declarations>

TYPE
    <type-declarations>

CONTEXT
    <context-var-declarations>

<procedure-declarations>

END
```

Each procedure declaration is introduced by the keyword `PROCEDURE`. Comments are started with `(*` and ended with `*)`.

The SCIDDLE *stub compiler* generates some header files and the client and server stubs. They contain the client and server side communication code, which must be compiled and linked with the client program and the server routines, respectively. Figure 3 shows the modules of a SCIDDLE application.

The following sections describe the interface definition language and explain how to use the stub compiler and how to call remote procedures from a client program.

4.1 Constant Declarations

Constant declarations are primarily used to define array dimensions and other constants shared by client and server. Constants are placed in the common header file and can be referenced by C programs. A constant declaration appears in the constant declaration section and has the form

```
<const-ident> = <const-value>;
```

where `<const-value>` can be either an integer value or the name of an other constant. Other values or expressions are currently not supported.

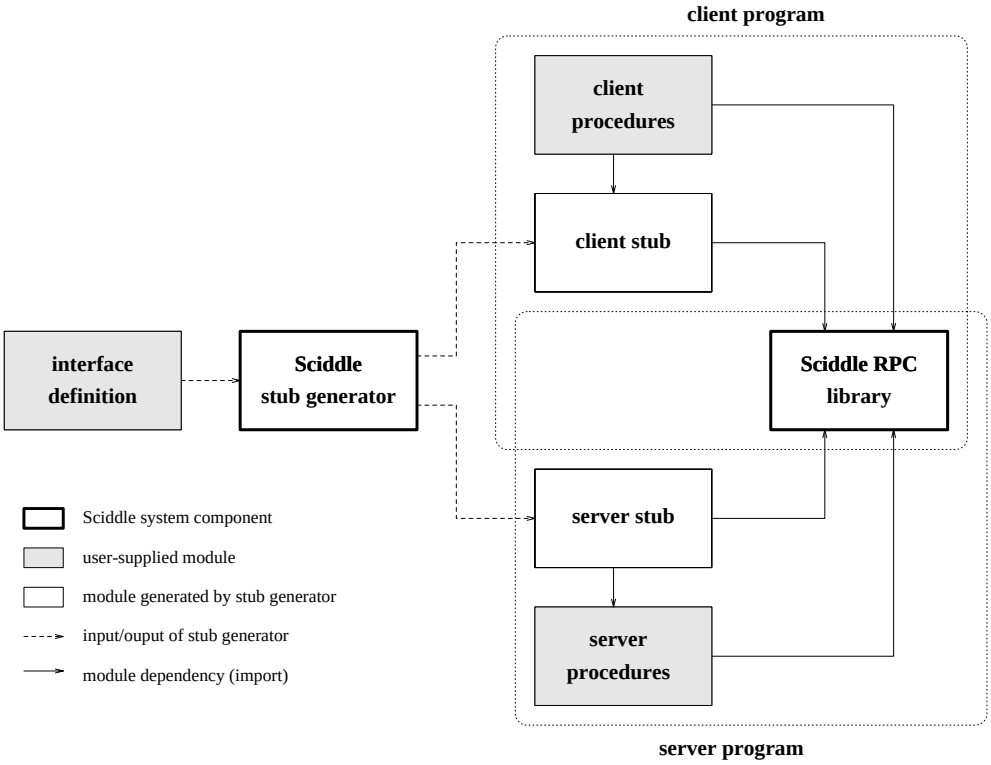

Figure 3: Components of a Sciddle application

4.2 Type Definitions

The following basic types are supported:

- short and long integers (`SHORTINT`, `INTEGER` and `LONGINT`)
- single and double precision floating point numbers (`REAL` and `LONGREAL`),
- single and double precision complex quantities (`COMPLEX` and `LONGCPLX`),
- characters (`CHAR`) and
- static and dynamic strings (`STRING` and `STRING[<length>]`).

Arbitrary array and record types may be defined from these basic types. A type definition appears in the type declaration section and has the form

```
<type-ident> = <type>;
```

Record definitions are surrounded by the keywords `RECORD` and `END`. Since record types are not supported by Fortran 77, no Fortran stubs can be generated from interface definitions containing record types.

Arrays may have up to 16 dimensions. An array has two attributes associated with it: a packing specification and a storage class. Packing refers to the index

space of an array, the storage class denotes whether the array is static (fixed size, allocated globally or on the stack) or dynamic (variable size, allocated on the heap). Arrays, subarrays and array attributes are described in more detail in section 4.5.

Variant records (or unions) and pointer types are currently not supported.

Type definitions are translated by the stub generator to C `typedefs`. They are located in the header files. The user-defined types can be used by C programs to declare variables and parameters. Their names are the same as the ones in the interface definition.

4.3 Context Variable Declarations

Context variables provide a mean of transferring data to the static data area of a server. Context variables are global variables of the server and thus persist across remote calls[3]. Context variables allow the client to update and read server context. They were introduced for efficiency, avoiding to copy (local) parameter data to static data space. A context variable declaration appears in the context section and has the following form:

```
<context-var-name> : <context-var-type>;
```

A context variable may have any of the predefined or user defined types. There is an additional type for context variables. The block type is very similar to a record type, but is surrounded by the keywords `BLOCK` and `END`. The major difference to records is that context block types must not contain any dynamic data and they are not allowed to appear in type definitions. But in contrast to records, context block types are supported by Fortran *and* C. In Fortran, they are mapped to COMMON blocks, in C to structures.

Context variables are updated or read by passing context parameters to remote procedures. Context parameters are described in the next section. In order to work correctly and in a portable way, context variables have to be installed on the server side. The way to do that is described in section 5 on the runtime library user interface.

4.4 Procedure Declarations

Procedures come in two flavors: synchronous and asynchronous. The keyword `SYNC` or `ASYNC`, appended to the parameter list and separated from it by a colon, specifies the flavor. Default is synchronous. The stub compiler generates one (client) stub for each synchronous and two stubs for each asynchronous procedure.

Each procedure parameter has a well-defined data type and is tagged with a direction attribute, which is either `IN`, `INOUT` or `OUT`. Parameters are copied over the communication channel in the respective directions. Parameter types may be any of the predefined basic types or user-defined types.

Procedure declarations appear in the procedure section and have the form:

[3]They were called context variables and not global variables in order to avoid confusion with global variables in a distributed sense which are shared by several servers and kept consistent by an appropriate protocol.

```
PROCEDURE <proc-name> ( <parameter-list> ) [ : <flavor> ] ;
```

There is a special parameter type to pass context variables. Defining a parameter type as `^CONTEXT` means that the parameter will be transferred to/from the server's context depending on its direction attribute. Such a parameter is called a *context parameter*. A context variable with the same name must be defined in the context section. Ordinary and context parameters may be arbitrarily mixed.

A description of how remote procedures are called is given in section 4.7.

Exceptions

For each procedure the user may define a number of *exceptions*. Exceptions are application level server procedure failures. They are declared in the interface definition just after the procedure head. The following is an example declaration of a procedure with two exceptions

```
PROCEDURE Solve(IN  n: INTEGER;
                IN  A: Matrix; IN  y: Vector;
                OUT L: Matrix; OUT x: Vector);
EXCEPTION BadValue, IllCondition;
```

Exceptions are used by the server to quickly signal the occurrence of a failure to the client. They can be raised by calling the user library routine `sc_exception()` with an exception code as an argument (see also section 5.3). No result values, but only the exception code is sent back in that case. The RPC then returns with the error code `RPC_FAILED` and the global RTS variable `sc_rpcret` contains the exception code indicating the type of failure (see also section 4.7).

Exceptions are numbered from 1 to 31 in the order of declaration. The common header file contains symbolic constants for the exception codes.

4.5 Array Handling

SCIDDLE provides means to distribute arrays to servers. In order to avoid communication overhead due to transmission of unneeded array elements, we allow partial arrays or subarrays to be selected for transmission. For this purpose, we have introduced *views*. Views are very similar to array sections in Matlab. A view is a regular subarray selection, composed of a set of triples of the general form `base-index:stride:end-index`, one for each array dimension. The index range of a view for dimension d starts at 1 and ends at n_d, where n_d is the number of elements in dimension d. Actually, there are four forms of views. They are listed in table 2.

In order to ensure that views are within the dimensions defined for the array, a range check is performed prior to transmission. Figure 4 (a) shows a two-dimensional array with the elements selected by the view `[2:2:4,1:2:5]`. Subarrays may be passed to servers by attaching a view to an array parameter in the parameter list of a procedure. Each component of a view is either a (numeric or symbolic) constant or an integer parameter appearing in the parameter list before the array parameter. Integer parameters passed as view components are called *bound* parameters.

view form	meaning
`base-index:stride:end-index`	base to end index with stride
`base-index:end-index`	base to end index
`end-index`	1 to end index
:	1 to n_d

Table 2: Different forms of views

Note: Bound parameters must have the `IN` direction attribute independent of the direction of the array parameter they are bound to, since the subarray is selected on the client side and once a selection is made, it is fixed.

As an example, look at the following procedure declaration.

```
PROCEDURE HorizontalStripe(IN    lo, hi:     INTEGER;
                           INOUT a[lo:hi,:]: ARRAY [M,N] OF REAL);
```

Parameter `a` is declared as a M-by-N array of floating point numbers. The elements of `a` actually passed in a call are selected by the two integer parameters `lo` and `hi`, indicating the start and end indices of a horizontal stripe of the array. The second view component ':' selects all columns of `a`.

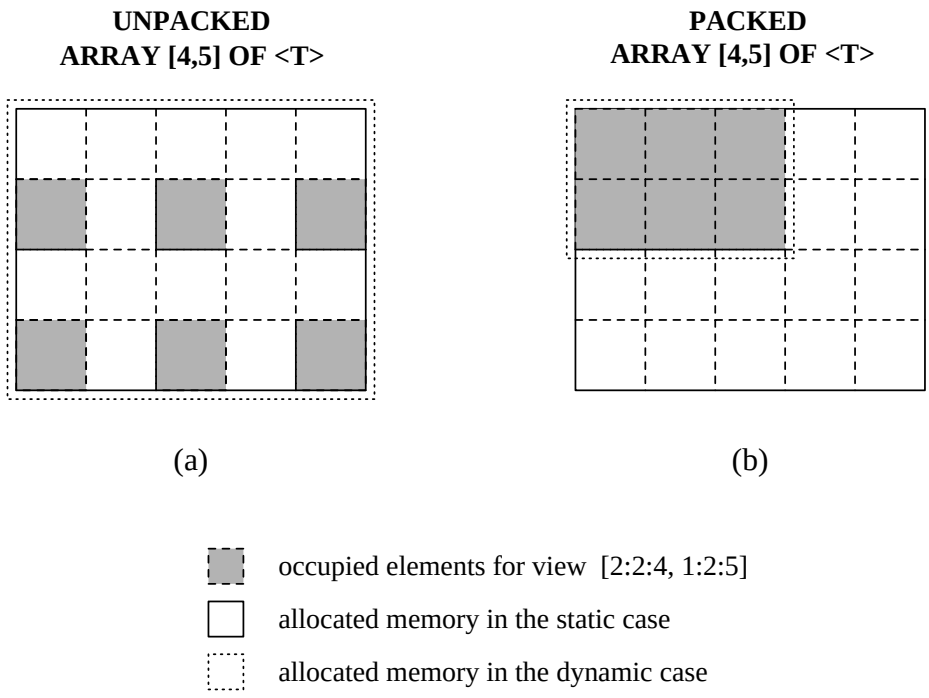

Figure 4: Subarray selection by views and packing

Reduction of network traffic is only one aspect of better resource utilization. We should also reduce the amount of memory occupied by subarrays of large arrays on

the server side. Therefore, array types have *packing* and *storage class* attributes as already mentioned above.

Packing (`PACKED` or `UNPACKED`) refers to the index space of arrays. An unpacked subarray keeps the index space of the original array, whereas the index space of a packed subarray is transformed such that all (occupied) elements are shifted towards the index "zero" point (1, 1, ..), i.e. base and stride are set to one in all dimensions. Packing does only affect subarrays but not full arrays.

The storage class may be either `STATIC` or `DYNAMIC`. A static array is fixed in size and thus occupies memory for the full size array according to its declaration. A dynamic array is variable in size and the allocation mechanism tries to reduce the amount of allocated memory to the space needed by the actual elements.

The effect of the packing attribute is illustrated in Figure 4 (a) and (b). The figure also shows the memory allocated for the array on the server side, depending on the attributes. Most economical with respect to memory space are packed, dynamic arrays.

The default attributes are static and unpacked.

4.6 Running the Stub Compiler

Note that by declaring the set of procedures, their parameter types and direction attributes, the application level communication protocol is completely defined. The SCIDDLE stub generator is therefore able to produce all the communication code necessary to transmit parameters from client to server and vice versa. The stub compiler is invoked by typing

```
% stubgen <options> <interface-definition-file>[.sc]
```

where the extension '`sc`' may be omitted. The compiler recognizes the following options:

- -f: generate stubs interfacing with Fortran client and server modules
- -c [c|fort] [row|col]: client options, client stub interfaces with C/Fortran client, arrays are stored in row/column major order in client
- -s [c|fort] [row|col]: server options, analogous to client options

The language switch makes the compiler generate the correct interface for the chosen language and adapt to the different naming conventions and array memory layouts. Fortran uses column major and C uses row major array representation. The user can choose a different order from the one given by the language. This feature is useful, if an existing mixed-language program already takes the reversed order of the other language into account. So the parts written in one of the languages will interpret arrays in a different than their native order. The switches tell the stub generator in which order the stubs should interpret array elements without changing type definitions.

The stub compiler emits the following message in response to successful processing of an interface definition:

```
% stubgen <interface-definition-file>[.sc]
Sciddle Stub Generator V3.0a
- <interface-definition-file>.sc
+ <interface-name>.h
+ <interface-name>_clt.h
+ <interface-name>_clt.c
+ <interface-name>_srv.h
+ <interface-name>_srv.c
No errors
```

Where `<interface-name>` is the name of the interface, i.e. the identifier appearing after the keyword `INTERFACE`. The stub generator produces five output files: a common header file, shared by client and server, two separate client and server headers and the client and server stub modules. Include the client header file in your client program and the server header file in the server program. The stub code is generated in C. If the appropriate options are set, the code can be called from Fortran programs. The naming conventions for stub routines will be presented in the following section.

The stub compiler puts a time stamp into both stubs which is cross-checked when the client connects to a server. This serves as a simple version check. On a version mismatch, an error code is returned and the connection is closed. This can be viewed as a global, per-interface, type check.

4.7 Calling Remote Procedures

Calling a remote procedure is very much like calling a local one with two minor differences: a remote procedure has a name prefix and its last parameter is an additional server identifier. Server identifiers are returned by the server startup routines in the user library. We will meet them in section 5. A complete stub routines reference is given in appendix C.

In C, parameters declared as `IN` are passed by value, whereas `INOUT` and `OUT` parameters are passed by reference. In Fortran 77, all parameters are passed by reference by definition of the language.

Synchronous Procedures

Assume, a procedure is declared as

```
PROCEDURE example(IN i: INTEGER; OUT r: REAL; INOUT c: CHAR);
```

This procedure is called from a C client as follows:

```
rc = call_example(i, &r, &c, sid);
```

where the parameter `sid` is a server identifier. The call returns a status code `rc`. A value of `SC_OK` indicates success, any value < 0 failure. In the case of failure, the value of `rc` is also found in the RTS error variable `sc_errno`. If the failure occurred

on the server side, the return code is SC_RPCFAILED and the RTS status variable sc_rpcret contains additional information about the reason of the (remote) failure. Table 3 shows the possible return conditions and the corresponding status codes in rc and sc_rpcret. Error codes and messages are listed in appendix E.

condition	return code rc	sc_rpcret
success	SC_OK	SC_OK
local failure	error code ($\neq$ SC_RPCFAILED)	undefined
remote failure	SC_RPCFAILED	remote value of sc_errno
exception	SC_RPCFAILED	exception code ($0 < exc < 32$)

Table 3: RPC return conditions

From a Fortran client the same procedure would be called like this

```
call callfexample(i, r, c, sid, rc)
```

Note that the prefix is callf instead of call_ and the condition code is returned as last parameter after the server identifier.

Asynchronous Procedures

Assume now, that the above procedure was declared asynchronous (by appending ':ASYNC' after the parameter list). Then the stub compiler would generate two client stub procedures, an *invoke* and a *claim* stub. The invoke stub sends the IN and INOUT parameters to the server and then returns to the caller, whereas the claim stub claims the result values, the INOUT and OUT parameters. The invoke stub takes a server identifier as its last parameter and returns a *call identifier* on success, which is a handle for the on-going call. This call identifier is consumed by the claim stub. Call identifiers are useful for matching invocations with results: the claim stub will only return those results associated with the call identifier.

In our example, this pair of calls looks like the following in C

```
cid = invoke_example(i, &c, sid);
rc = claim_example(&r, &c, cid);
```

Again, if the invocation or claim fails, an error code is returned. Note that a 'remote failure' or 'exception' condition (see above) can only occur with the latter. In Fortran, the same sequence appears as

```
call invokefexample(i, c, sid, cid);
call claimfexample(r, c, cid, rc);
```

In contrast to the C version, the prefixes invoke_ and claim_ are replaced by invokef and claimf and the C return values are returned in the last (additional) parameter.

Note: The actual server procedure has exactly the name and parameters declared in the interface module independent of whether the procedure is synchronous or asynchronous. No change is necessary on the server side, to turn a local into a remote procedure.

Procedures with Context Parameters

Context parameters are only passed on the client side, but they are not passed to the actual server procedure by the server stub, since they are transferred directly to or from the server context. Let us consider an example. The declaration

```
PROCEDURE Blu(IN a: INTEGER; OUT b: ^CONTEXT; INOUT s: STRING);
```

where `b` is a context variable of type, say, `REAL`, states that `Blu` is a procedure which takes an integer and a string as arguments and returns a floating point value from the server context and a string. The procedure is called from the client like this

```
rc = call_Blu(a, &b, s);
```

In contrast, the context parameter `b` does not appear on the parameter list of the server procedure. Here is its prototype

```
void Blu(sc_integer a, sc_string s);
```

Note: If there are exclusively context and bound variables in a procedure declaration, no server procedure will be called by the server stub. The procedure is then considered as a pure context transfer procedure.

5 Runtime Library User Interface

This section describes the functions available to the user through the runtime library user interface. It mainly includes routines for service and server management and for management of on-going asynchronous calls. You can find a complete user library reference in appendix D. The name of the library is `libsc.a`. If you want to call these routines from a Fortran program, you also need the interface conversion library `libf2c.a`. Note that the Fortran routines have the prefix `scf` instead of `sc_` and that return values are passed back in the (additional) last parameter.

5.1 Service and Server Management

Before a service can be used in a client program, it needs to be attached. Attaching a service initializes the client stub and registers the service with the runtime support. The routine to do that is

```
int rc =  sc_svcattach(char *svcname, void (*svcinit)());
```

The first parameter is a string typed identifier to be used for that service. The second is an initialization routine named `init_<interface-name>()` exported by the client stub (see also appendix C).

When a service is no longer needed, you may want to detach it from the client. You can do that by calling the detach routine

```
int rc = sc_svcdetach(char *svcname);
```

or you may detach all services at once with

```
int rc = sc_svcdetachall(void);
```

Detaching a service involves termination of all servers running the service. All these routines return SC_OK on success and a negative error code on failure. See appendix E for error codes.

SCIDDLE servers are started by their clients and are private to them, i.e. a server can be accessed only by its (single) client. Also, their lifetime is bound by the lifetime of the client. The basic function to start up a server is

```
int rc = sc_srvstart(char *host, char *svcname, int niceval);
```

This initiates an instance of the service `svcname` on the indicated host. The last parameter `niceval` is a priority value. The higher it is, the lower the priority of the server on the target host. It returns a server identifier on success and a negative error code on failure. The server identifier is used to issue calls to servers as we have seen already in section 4.7.

There are two routines which start up multiple servers at once:

```
int rc = sc_srvstartmulti(char *host, char *svcname, int niceval,
                          int numinst, int *srvids);

int rc = sc_srvstartconfig(char *configfile, int niceval,
                           int *numinst, int *srvids);
```

The first, `sc_srvstartmulti()`, starts up the indicated number of instances on the server host and the second, `sc_srvstartconfig()` reads the services and hosts from a configuration file. The file has the following format (see also appendix F for file formats):

```
<host-name>   <#instances>   <service-name>
```

Both routines return an array of server identifiers. Their return value is `SC_OK` on success and a negative error code on failure.

You can terminate a server by calling the following routine with the server identifier as parameter:

```
int rc = sc_srvterm(int sid);
```

It is also possible to send arbitrary UNIX signals to servers. The routine to do that is

```
int rc = sc_srvsendsig(int sid, int signo);
```

where `signo` is a signal number as defined in the system header file `signal.h` (see also `kill(2)` and `signal(3)`).

Finally, there is a routine that returns the name of the server host for a given server identifier:

```
int rc = sc_srvgethost(int sid, char *host);
```

Next, we consider the routines for call management.

5.2 Call Management

Call management refers to the handling of on-going asynchronous RPCs. Recall that an asynchronous call is split into two parts on the client side: invoking the procedure and claiming the results. The corresponding (client) stub procedures are called the *invoke* and the *claim* stub, respectively. Calling the invoke stub issues a RPC and returns a *call identifier* (see also section 4.7). Call identifiers are consumed by the claim stubs and serve to match results with invocations, i.e. the claim stub will only return the results associated with the invocation which produced it. Calling a claim stub which does not match the invoke stub will return an error. Call identifiers are also a handle for applying several operations. You may want to know whether a call has already been completed. You can do so by calling

```
int rc = sc_callisready(int cid, long timeout)
```

The first parameter is a valid call identifier returned by a successful call to an invoke stub and the second parameter is a timeout value in milliseconds. Specifying zero for `timeout` polls for readiness, specifying -1 is equivalent with an infinite timeout and blocks.

If you are no longer interested in the result of a call, you can discard the result values. The routine

```
int rc = sc_calldiscard(int cid)
```

discards a call identifier and the results associated with it.

The following two routines get some more information about a call.

```
int procid = sc_callgetprocid(int cid)
int srvid  = sc_callgetsrvid(int cid)
```

The first returns the ID of the procedure that was called. Procedure identifiers are defined in the common header file. The second returns the ID of the server that was called.

Call Groups

Call groups are an effective way to deal with a set of on-going asynchronous calls. Call identifiers can be added to a call group. The following are the routines to create and destroy call groups, get the size of a call group and add a call identifier to a call group:

```
int cgid = sc_cgcreate(void)
void     sc_cgdestroy(int cgid)
int rc   = sc_cgadd(int cgid, int cid)
int size = sc_cggetsize(int cgid)
```

The following routine returns the first call identifier in the group to become ready.

```
int cid = sc_cgextractready(int cgid)
```

It blocks, if there are no results immediately available. The returned call identifier is excluded from the call group.

Finally, there is a routine that discards all the call identifiers in a call group:

```
int rc = sc_cgdiscardall(int cgid)
```

A typical application of call groups looks is to you issue several asynchronous calls and include the resulting call identifiers into a call group. Later, the routine `sc_cgextractready()` is repeatedly called on the group and the ready call IDs are claimed until the group becomes empty. A corresponding program fragment is shown below

```
cgid = sc_cgcreate();

for (p = 0; p < nserv; p++)       /* issue the calls */
{
    new_cid = invoke_<some-proc>(<some-args>, sid[p]);
    rc = sc_cgadd(cgid, new_cid);
}

while (sc_cggetsize(cgid) > 0)    /* claim results */
{
    rdy_cid = sc_cgectractready(cgid);
    rc = claim_<some-proc>(<some-results>, rdy_cid);
}

sc_cgdestroy(cgid);
```

This construct allows to claim the results of several asynchronous calls in the order they become ready. As a whole, it can be viewed as a parallel RPC.

5.3 Miscellaneous Functions

Installing Context Variables

In order to use context variables you need to tell the runtime system where they are located. For this purpose, call the routine

```
int rc = sc_installcontext(int num, char *cvp0, char *cvp1, ...)
```

and pass it the number of context variables and their addresses[4]. Do this in the initialization section of the server program. The best place is just after the call to the setup stub routine. The number and order must be the same as in the interface definition. The Fortran interface for this routines is a bit different than usual, see appendix D.

Raising Exceptions

A server procedure can signal the occurrence of a failure to the client by raising an *exception*. The function is

```
int rc = sc_exception(int excptcode)
```

It returns the exception code `excptcode` to the client instead of the normal results. Exceptions can be declared along with each procedure in the interface definition (see section 4.4).

[4]We admit that this quite a bad construct, but it is portable across all platforms and that is the reason we chose it.

Array Memory Allocation

There are two routines to allocate and free memory for multi-dimensional arrays in C programs:

```
int rc = sc_allocarray(char **ap, int ndims, int *dims, int elsiz)
void     sc_freearray(char *a)
```

The first, `sc_allocarray()`, takes a generic pointer to an array. The other parameters specify the array dimensions and the size of a single element. An array may have up to 16 dimensions. An array allocated with this routine can be freed with `sc_freearray()`.

Message Buffer Size

The message buffer sizes can be adapted to the needs of the application by calling

```
int rc = sc_setbufsize(size, which)
```

where `which` selects the buffer whose size is to be changed. Zero (0) means the receive, one (1) the send buffer and two (2) means both buffers. In order to transfer data efficiently, message buffers should not be too small compared to the size of messages that are transmitted. The default size for both buffers is 16 KBytes.

5.4 Error Codes and Messages

The SCIDDLE RTS maintains two global status variables: `sc_errno` and `sc_rpcret`. `sc_errno` contains the error code of the last library routine that failed. When a RPC fails due to a remote RTS error or an exception, `sc_errno` contains the error code `SC_RPCFAILED` and `sc_rpcret` the remote error code or exception code respectively. Error codes are negative, exception codes positive values between 1 and 31 (see also section 4.7). A zero return value (`SC_OK`) indicates success. See appendix E for a complete list of the RTS error codes and messages.

All RTS errors are soft errors by default, which means the routines which produced the error set the error variable `sc_errno` and return the error code to the caller. Sometimes, in particular during debugging, it is useful to terminate the program when an error occurs. The routine

```
void sc_softerr(int onoff)
```

allows to switch between soft and hard errors. The parameter is a boolean value.

There is a procedure which allows to print the error message associated with the error code in `sc_errno`. It works just like the UNIX `perror()` routine and is called as shown below.

```
void sc_perror(char *msg)
```

By default, error messages are printed automatically when an error occurs. This can be turned on and off with

```
void sc_autoerr(int onoff)
```

The parameter is a boolean value.

6 The Monitor

The primary task of the SCIDDLE monitor is to collect the terminal output (stdout and stderr) of all servers and display it on a single terminal. Also, you will be prompted for passwords in the monitor's terminal if requested by the super-server. In addition, there are some commands which are interpreted by the monitor. They serve to start, check and stop super-servers.

When starting the monitor a message like the following is printed

```
Sciddle Monitor 1.0
listening on default port
ready, online...
```

On the second line the monitor prints the port where it is listening for connection requests from the network. In the shown case it is the default port. If the default port is already occupied, the monitor takes the next free port and prints out its number.

6.1 Display Mode

When you start the monitor, it is in online or display mode. This is indicated by the message `ready, online...`. In this mode the monitor collects the terminal output from application servers.

In order for this to work the master (top-level) client has to know the monitor address (i.e. machine and port number) when the application is started. By default, the monitor is assumed to run on the same machine as the master and to listen on the default port. If machine and/or port differ from the default, they need to be supplied to the master with the option `-m <machine> <port>`. Either argument is replaced by the default if omitted.

If you forgot the monitor port number, you can display it by typing `port` in the monitor command mode (see below). Make sure that the monitor is in display mode when you are running an application.

6.2 Command Mode

If you type `<return>` the monitor switches to command mode and prompts you for a command line with the hash mark (`#`). The following commands are available:

- `start` - start a super-server[5]
- `ping` - ping a super-server for readiness
- `stop` - stop a super-server
- `port` - prints the monitor port

[5] does only work if the specified host permits to execute commands with `rsh`, make sure `rsh` works for that host and that the local host is listed in the remote `.rhosts` file (see `rsh(1)` or `remsh(1)`) otherwise you need to login and start it "by hand"

- `reconfig` - reload the host configuration file `~/.sc_hosts`
- `!` - escape character for shell commands
- `?/help` - print a help message
- `quit` - leave the monitor

The first three commands take a mandatory machine name argument and an optional login and port argument. The login is the user name on that machine and the port is the the super-server port. The monitor loads the host configuration file `.sc_hosts` in your home directory while starting. The logins and ports listed there become the default values. If you omit the login and port and there is no entry in the host file, the login is assumed to be the same as on the local machine and the port is assumed to be the default port for the super-server. The `start` command also understands the `-s` and `-l` options for the super-server (see section 2.4).

After having processed a command the monitor automatically switches back to display mode (it prints the '`ready, online...`' message). In order to enter more commands type `<return>` again.

7 Writing and Building your Applications

Recall the components of a SCIDDLE application: there are client and server modules and an interface definition for each service the application claims (see figure 3). Writing and building a SCIDDLE application usually involves the following steps:

1. partition your application into client and server components
2. define a remote interface definition for each service
3. implement the client and server modules
4. distribute the sources along with the interface definition(s) to all machines where you want to run your application
5. compile the interface definition(s) with the stub compiler `stubgen`
6. compile client and server modules as well as all stubs and link them together with the runtime system/user library **libsc.a** to form the client and server programs; if one of your components is written in Fortran also link the interface conversion library **libf2c.a**

Note the following:

- *Ad 4 and 5*. The stub compiler puts a time stamp for run-time version check into the stub modules. This time stamp is derived from the last modification date of the interface definition. This date might get modified when copying files with `ftp`. Creating a tape archive for the purpose of remote copying prevents the last modification date from being changed. Alternatively, you can also copy the stubs and headers.

- *Ad 6*. There is a prototype makefile called `AppMakefileProto` in the SCIDDLE base directory. You can use this one for simple applications with only one server program. The only things you have to change are the architecture tag and machine specific compiler/loader flags. Note that the architecture tag should be defined for the C preprocessor with `-D<ARCH>` when compiling the stub modules. Machine specific compiler loader flags are listed in table 4 below.

ARCH	Flag	Meaning
SGI	`-cckr`	(cc compiler) sets K&R C
	`-lsun`	(loader) links Sun libray
CRAY	`-dp`	(cf77 compiler) float format
SX3	`-hfloat1`	(cc compiler/loader) float format
	`-float1`	(f77sx compiler/loader) float format

Table 4: Machine specific compiler/loader switches

8 Running your Applications

There are a couple of things to consider before you start your application. These are

1. the super-servers must be set up correctly on the server machines
2. there must be an entry for all services in the services file on the server machines
3. you have to start a monitor that collects the server output
4. the client needs to know the monitor address

These items will be discussed now.

8.1 Checking the Super-Server Setup

Make sure that there is a super-server running on each of the server machines you want to use and that the host configuration file `.sc_hosts` in your home directory lists the correct ports and login names. You can use the monitor's `ping` command to check the super-servers for readiness.

8.2 The Services File `.sc_services`

Before you run your application, you have to make an entry in the services file `.sc_services` in your home directory on each server machine for each of the services your application claims. The services file maps service names to executables.

Its entries have the form

```
<service-name>  <exec-path-name>
```

where the service name on the left hand side corresponds to the service name you assign when attaching the service to the client with `sc_svcattach()`. The executable path name is the path to the executable including its name. Relative paths start from your home directory. A percent sign (%) in the path is expanded to the architecture tag of your machine. On a Sun-4 architecture, for instance, the path

```
sciddle3.0/myapp/%/myappServer
```

would be expanded by the super-server to

```
<your-home-directory>/sciddle3.0/myapp/SUN4/myappServer
```

Note that this file must not be accessible by any other user, otherwise the super-server will refuse to start the server.

8.3 Running the Monitor

Start a SCIDDLE monitor. The monitor prints out the port number where it is listening. You need to know this port number in order to start the application correctly. If your monitor is already running and the port number has disappeared from the display, you can use the monitor's `port` command to show it. Make sure the monitor is online and not in command mode.

8.4 Starting the application

In order to start the application, simply type the name of the master client program on the command line. Additionally, you have to supply the address of the monitor, if

1. the monitor runs on a different machine than the master and/or
2. the monitor is not listening at the default port

In the former case you have to specify the machine name where the monitor runs and in the latter the port number where it listens with the option `-m`. For instance, if the master runs on `micky` and the monitor runs on `goofy` and is listening on port 7777, then start the master program `appclient` as follows

```
micky% appclient -m goofy 7777
```

9 Example: Parallel Distributed Matrix Multiplication

In this section, we consider as an example application parallel distributed matrix multiplication. We want to compute the matrix product $C = A * B$ in parallel on, say, p servers. For simplicity, we assume that the matrices are square of order n.

There are several ways to implement such an algorithm, depending on how the data is distributed on the servers. Figure 5 shows one way how this can be accomplished. Each processor computes a block of columns of C. To that end, it needs to know the corresponding columns of B and the complete A.

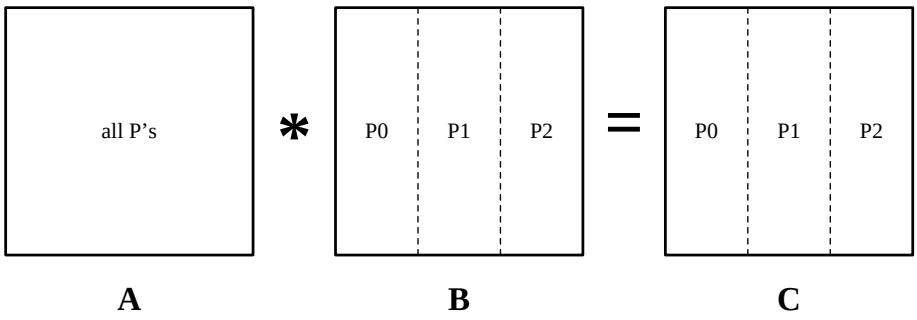

Figure 5: Decomposition of matrix multiplication into 3 independent subproblems

After these considerations, we can derive the *remote interface definition* for our parallel matrix multiply service. It looks like the following:

```
INTERFACE MatMult;
(* remote interface definition for parallel matrix multiply *)

CONST
    Max = 100;

TYPE
    Matrix = ARRAY [Max, Max] OF REAL;


PROCEDURE Multiply(IN  n, lo, hi            : INTEGER;
                   IN  A[n, n], B[n, lo:hi] : Matrix;
                   OUT C[n, lo:hi]          : Matrix): ASYNC;

END
```

Notice that the procedure `Multiply` is declared asynchronous (`ASYNC`), since we want to issue a number of calls simultaneously which compute the matrix product in parallel. Clearly, the matrices `A` and `B` must be declared as IN parameters and the result `C` as an OUT parameter. The arrays have a maximum size of `Max` $\times$ `Max`

or 100×100 elements. Since the actual problem size n is a command line argument of the client program, we constrain the actual size of the matrix `A` by the problem size `n`, such that no unused elements are transmitted to the servers. Only a vertical stripe of the matrices `B` and `C` is passed/returned in each call, so we set a column view which selects the columns of these matrices. The integer values `lo` and `hi` indicate the base and end index of these columns. Note that the bound parameters `n`, `lo` and `hi` must have the `IN` direction attribute.

The implementation of the server procedure is straightforward and is shown in Figure 6. Note that the limits of the inner for-loop are determined by `lo` and `hi`. Note also the off-by-one index shift which is necessary in C programs because view indices start at 1.

```
#include "MatMult_srv.h"

void Multiply(n, lo, hi, A, B, C)
int n, lo, hi;
Matrix A, B, C;
{
    int i, j, k;
    float sum;

    for (i = 0; i < n; i++)
    {
        for (j = lo-1; j < hi; j++)   /* shift j=1 to j=0 */
        {
            sum = 0.0;
            for (k = 0; k < n; k++)
                sum += A[i][k] * B[k][j];

            C[i][j] = sum;
        }
    }
}
```

Figure 6: Server procedure for distributed matrix multiplication

The client program is slightly more complicated. It is shown in Figure 7. First, the master program is initialized with the routine `sc_initmaster()` and then the problem size and the number of (virtual) processors are read from the command line and matrices `A` and `B` are initialized. After that, the matrix multiply service is attached to the client by invoking the `sc_svcattach()` routine. The first parameter in this call is a string identification for the service and the second parameter is the initialization routine `init_MatMult` exported by the client stub. Then, a pool of servers, listed in the file `matmult.conf`, is started up by invoking the

sc_srvstartconfig() procedure. It returns the number of started servers along with their server IDs. The program stops execution, if it got less servers than requested.

```
#include "sciddle.h"
#include "MatMult_clt.h"

#define MAXSRV  32

main(argc, argv)
int argc;
char **argv;
{
    Matrix A, B, C;
    int n, k, p, nsrv, step, rest;
    int hi, lo, cgid, cid, sids[MAXSRV];

    /* init Sciddle master */
    sc_initmaster(&argc, argv);

    /* read problem size from command line */
    n = atoi(argv[1]);
    p = atoi(argv[2]);

    /* initialize matrices A and B */
    initMatrices(n, A, B);

    /* initialize the service and start the server config */
    nsrv = MAXSRV;
    sc_svcattach("MATMULT", init_MatMult);
    sc_srvstartconfig("matmult.conf", 0, &nsrv, sids);

    if (nsrv < p) exit(nsrv);
```

Figure 7: Client program for distributed matrix multiplication

```
    /* distribute work evenly among p servers */
    step = n/p; rest = n%p;
    if (rest) step++;
    lo = 1; hi = step;

    cgid = sc_cgcreate();    /* create a call group */

    for (k = 0; k < p; k++)
    {
        cid = invoke_Multiply(n, lo, hi, A, B, sids[k]);
        sc_cgadd(cgid, cid);

        lo += step;
        hi += (k == rest-1) ? --step : step;
    }

    /* wait for the results */
    while (sc_cggetsize(cgid) > 0)
    {
        cid = sc_cgextractready(cgid);
        claim_Multiply(C, cid);
    }
    sc_cgdestroy(cgid);      /* destroy call group */

    /* detach service (terminates servers) */
    sc_svcdetach("MATMULT");
}
```

Figure 7: Client program for distributed matrix multiplication (end)

After these initialization and setup actions we now come to the interesting part of the program. The work needs to be distributed to the servers. We first compute the width of the vertical stripes in `step` and assign initial values to the index boundaries `lo` and `hi`. Note again that views indices start at 1, though C indices start at 0. Then, we create a call group to which we add the call identifiers (with `sc_cgadd()`) that are produced by the server invocations (`invoke_Multiply()`). After each call the index boundaries are shifted by the correct amount to the right.

After all processors have been assigned their work, we have nothing to do but to wait for the results. As long as there are still call IDs in the group we call `sc_cgextractready()` which possibly blocks and eventually returns the first call ID to become ready. The ready ID (`cid`) is excluded from the group and the results are claimed by calling the stub routine `claim_Multiply()` with `cid` as a parameter.

Finally, the call group is destroyed and the service `MATMULT` detached which causes all `MATMULT` servers to be terminated.

A Interface Definition Language Syntax

This appendix covers the syntax of the SCIDDLE remote interface definition language in BNF notation. Terminal symbols are quoted. `epsilon` denotes the empty production ϵ.

```
Interface       ::= 'INTERFACE' InterfaceHead InterfaceBody 'END'
InterfaceHead   ::= InterfaceName ';'
InterfaceName   ::= IDENT
InterfaceBody   ::= Constants Types ContextVars Procedures
Constants       ::= 'CONST' ConstDeclList
                  | epsilon
ConstDeclList   ::= ConstDecl
                  | ConstDeclList ConstDecl
ConstDecl       ::= ConstName '=' Const ';'
Const           ::= ConstName
                  | NUMBER
ConstName       ::= IDENT
Types           ::= 'TYPE' TypeDeclList
                  | epsilon
TypeDeclList    ::= TypeDecl
                  | TypeDeclList TypeDecl
TypeDecl        ::= TypeName '=' Type ';'
TypeName        ::= IDENT
Type            ::= UserDefinedType
                  | BuiltInType
                  | ComposedType
UserDefinedType ::= TypeName
BuiltInType     ::= 'SHORTINT'
                  | 'INTEGER'
                  | 'LONGINT'
                  | 'REAL'
                  | 'LONGREAL'
                  | 'COMPLEX'
                  | 'LONGCPLX'
                  | 'CHAR'
ComposedType    ::= ArrayType
                  | RecordType
                  | StringType
StringType      ::= 'STRING' StringDimension
StringDimension ::= '[' Dimension ']'
                  | epsilon
ArrayType       ::= ArrayAttrs 'ARRAY' '[' DimensionList ']' 'OF' Type
ArrayAttrs      ::= Packing StorageClass
Packing         ::= 'PACKED'
                  | 'UNPACKED'
                  | epsilon
StorageClass    ::= 'STATIC'
                  | 'DYNAMIC'
```

```
                  | epsilon
DimensionList   ::= Dimension
                  | DimensionList ',' Dimension
Dimension       ::= Const
                  | '*'
RecordType      ::= 'RECORD' FieldDeclList 'END'
FieldDeclList   ::= FieldDecl
                  | FieldDeclList FieldDecl
FieldDecl       ::= FieldNameList ':' Type ';'
FieldNameList   ::= FieldName
                  | FieldNameList ',' FieldName
FieldName       ::= IDENT
ContextVars     ::= 'CONTEXT' ContextDeclList
                  | epsilon
ContextDeclList ::= ContextVarDecl
                  | ContextDeclList ContextVarDecl
ContextVarDecl  ::= ContextNameList ':' ContextVarType ';'
ContextNameList ::= ContextVarName
                  | ContextNameList ',' ContextVarName
ContextVarName  ::= IDENT
ContextVarType  ::= RegularType
                  | BlockType
RegularType     ::= Type
BlockType       ::= BLOCK FieldDeclList END
Procedures      ::= epsilon
                  | Procedures ProcedureDecl
ProcedureDecl   ::= ProcHeadDecl ExceptionDecl
ProcHeadDecl    ::= PROCEDURE ProcName ParameterList ProcType ';'
ProcType        ::= ':' SYNC
                  | ':' ASYNC
                  | epsilon
ProcName        ::= IDENT
ParameterList   ::= '(' ParamDeclList ')'
                  | epsilon
ParamDeclList   ::= ParamDecl
                  | ParamDeclList ';' ParamDecl
ParamDecl       ::= Direction ParamNameList ':' ParamType
Direction       ::= 'IN'
                  | 'OUT'
                  | 'INOUT'
                  | epsilon
ParamNameList   ::= ParamNameAnView
                  | ParamNameList ',' ParamNameAnView
ParamNameAnView ::= ParamName ParamView
ParamName       ::= IDENT
ParamView       ::= '[' ViewList ']'
                  | epsilon
ViewList        ::= View
                  | ViewList ',' View
```

```
View            ::= ViewComponent ':' ViewComponent ':' ViewComponent
                  | ViewComponent ':' ViewComponent
                  | ViewComponent
                  | ':'
ViewComponent   ::= IDENT
                  | NUMBER
ParamType       ::= RegularType
                  | ContxtParamType
                  ;
ContxtParamType ::= '^' CONTEXT
ExceptionDecl   ::= epsilon
                  | EXCEPTION ExceptNameList ';'
ExceptNameList  ::= ExceptionName
                  | ExceptNameList ',' ExceptionName
ExceptionName   ::= IDENT
```

B Stub Compiler Error Messages

Scanner Errors

- illegal token: <token>

Syntax Errors

- syntax error
- semicolon expected
- END expected
- direction attribute expected

Semantic Errors

- identifier <id> not declared
- identifier <id> redefined
- undeclared identifier <id> in view
- context variable <id> not declared
- dynamic data inside context block type
- illegal view for non-array parameter
- illegal zero stride <str> in view
- constant view component <vcmp> out of range
- array and view must have same number of dimensions
- view component <vcmp> must be an integer or constant
- view component <vcmp> must have IN direction
- dynamic strings not supported by Fortran
- array type not supported by Fortran
- records are not supported by Fortran

Other Errors

- compiler restriction: too many exceptions (31 allowed)
- compiler restriction: 16 dims max
- compiler internal error
- not implemented

C Stub Reference

This appendix describes the procedures generated by the SCIDDLE stub compiler and exported by the stub modules.

The reference uses generic names for the interface and declared procedures. The interface is called `interface` and procedures are called `proc`.

call_proc() callfproc()

Call synchronous remote procedure `proc()`.

Synopsis

C	`int rc = call_proc(<args>, sid)`
Fortran	`call callfproc(<args>, sid, rc)`

Parameters

`<args>`	parameters for procedure `proc` as declared in the interface definition
`sid`	id of server to call

Description

The stub `call_proc()` calls the remote procedure `proc()` on the server identified by `sid`. The call is executed synchronously, i.e. the caller (client) is blocked until the results arrive. The procedure may or may not be explicitly declared synchronous (`SYNC`) in the interface definition for the stub compiler to generate this stub, since synchronous is default.
All arguments, `IN`, `INOUT`, and `OUT`, as declared in the interface definition are passed to this routine. Note that the server id parameter is not passed to the remote procedure. The return value is an error indication. There are four types of return values:

1. The call has successfully completed and returns `SC_OK`.
2. An error occured on the client side returning an corresponding error code.
3. An error occurred on the server side, returning `SC_RPCFAILED`. The global variable `sc_rpcret` contains the remote error code (remote value of `sc_errno`.
4. The server procedure raised an exception, returning a (positive) exception code.

Error Codes

Code	Possible Reason
SC_NOSERVER	The server identification is invalid.
SC_BADPROC	Non-existent remote procedure called. The RPC protocol is out of sync.
SC_RPCFAILED	There was an error on the server side. The remote error code is contained in `sc_rpcret`.
SC_STRTOOLONG	You have passed a string that is longer than declared in the interface definition.
SC_BADVIEW	A view component parameter is out of range.
SC_BROKEN	The connection to the client is broken.
SC_BADMSG	Some data got scrambled. The stub cannot decode the RPC reply message. The RPC protocol is out of sync.
SC_NODATA	Read past end of message. The reply message contains less data than expected. The RPC protocol is out of sync.

invoke_proc() invokefproc()

Invoke remote procedure `proc` asynchronously.

Synopsis

C `int cid = invoke_proc(<in-args>, sid)`
Fortran `call invokefproc(<in-args>, sid, cid)`

Parameters

`<in-args>`	`IN` and `INOUT` arguments as declared for procedure `proc` in the interface definition
`sid`	id of server to call

Description

The stub `invoke_proc()` calls the remote procedure `proc()` on the server identified by `sid`. The call is executed asynchronously, i. e. the caller (client) is *not* blocked. The routine returns as soon as the request message has been sent off. The procedure must be explicitly declared asynchronous (`ASYNC`) in the interface definition for the stub compiler to generate this stub.
Only `IN` and `INOUT` parameters as declared in the interface definition, are passed to this routine. In C, `INOUT` parameters must be passed by reference. Note that the server id parameter is not passed to the remote procedure.
On success, a call identifier is returned. The call identifier must be passed later to the claim stub (see `claim_proc()` to get the results matching the current invocation.

Error Codes

Code	Possible Reason
SC_NOSERVER	The server identification is invalid.
SC_STRTOOLONG	You have passed a string that is longer than declared in the interface definition.
SC_BADVIEW	A view component parameter is out of range.
SC_BROKEN	The connection to the client is broken.

claim_proc() claimfproc()

Claim results of asynchronous invocation of remote procedure `proc`.

Synopsis

C `int rc = claim_proc(<out-args>, cid)`
Fortran `call callfproc(<out-args>, cid, rc)`

Parameters

`<out-args>`	`INOUT` and `OUT` parameters as declared in the interface definition for procedure `proc`
`cid`	call identifier from previous invocation

Description

The stub `claim_proc()` claims the results of a previous invocation of the remote procedure `proc`. Client and server are synchronized. The client may block until the results arrive. The procedure `proc` must be explicitly declared asynchronous (`ASYNC`) in the interface definition for the stub compiler to generate this stub. Only `INOUT`, and `OUT` arguments as declared in the interface definition are passed to this routine. In C all parameters have to be passed by reference. The return value is an error indication. There are four types of return values:

1. The call has successfully completed and returns `SC_OK`.
2. An error occurred on the client side returning an corresponding error code.
3. An error occurred on the server side, returning `SC_RPCFAILED`. The global variable `sc_rpcret` contains the remote error code (remote value of `sc_errno`.
4. The server procedure raised an exception, returning a (positive) exception code.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call identifier is invalid.
SC_BADPROC	Non-existent remote procedure called. The RPC protocol is out of sync.
SC_RPCFAILED	There was an error on the server side. The remote error code is contained in `sc_rpcret`.
SC_STRTOOLONG	You have passed a string that is longer than declared in the interface definition.
SC_BADVIEW	A view component parameter is out of range.
SC_BROKEN	The connection to the client is broken.
SC_BADMSG	Some data got scrambled, the stub cannot decode the RPC reply message. The RPC protocol is out of sync.
SC_NODATA	Read past end of message. The reply message contains less data than expected. The RPC protocol is out of sync.

init_interface() initfinterface()

Initialize a client stub.

Description

The stub compiler generates an initialization routine in each client stub. This routine must be passed to `sc_svcattach()` to initialize a service correctly (see user library reference). Synopsis, parameters and error conditions are not relevant here, since this stub is not called directly by your programs.

Note: When calling scfsvcattach() in Fortran, the init stub must be declared `external` or your program will probably crash.

Example

```
INTERFACE yourface;
(* interface for YOURFACE service *)
       .
       .
END


/* attach YOURFACE service */
rc = sc_svcattach("YOURFACE", init_yourface);

/* use it */
```

setup_interface() setupfinterface()

Set up a service.

Synopsis

C	`int rc = setup_interface(void)`
Fortran	`call setupfinterface(rc)`

Parameters

none

Description

The setup procedure is exported by the server stub. It must be called by the server program to setup the service. It initializes the server's runtime system and accepts a connection from the client.

Note: This is the SCIDDLE routine which should be called first in your server program. Do not call any other SCIDDLE routine before the set-up routine (except the "light-weight" routines sc_softerr(), `sc_autoerr()` and `sc_installcontext()`, which are harmless).

Example

```
INTERFACE yourface;
(* interface for YOURFACE service *)
       .
       .
END
```

```
/* minimal server program in C */
main()
{
    setup_yourface();
    run_yourface();
}
```

Error Codes

Code	Possible Reason
SC_NOHOSTFILE	Could not open host file `.sc_hosts` in your home directory.
SC_BADHOSTFILE	Format error in host file.
SC_NOHOST	A host listed in the host file is not known. The client host is not known to the server host.
SC_STUBVERS	Version mismatch between client and server stub.
SC_RTSVERS	Version mismatch between the client and server runtime system.
SC_BROKEN	The connection to the client is broken.
SC_TIMEOUT	The client did not connect to the server within the timeout period.
SC_TABOVFL	Host table overflow. Too many hosts listed in the hosts file.

run_interface() runfinterface()

Run a service.

Synopsis

C	`int rc = run_interface()`
Fortran	`call runfinterface(rc)`

Parameters

none

Description

This procedure runs a service. It must be called by the user from the server (main) program. It invokes the call dispatcher which repeatedly services RPC requests from the client. It does not return unless an error occurs. Note that the setup routine must be called prior to this routine.

Example

```
INTERFACE yourface;
(* interface for YOURFACE service *)
       .
       .
END
```

```
program server
c minimal server program in Fortran

integer rc

call setupfyourface(rc)
call runyourface(rc)
```

Error Codes

Code	Possible Reason
SC_NOSERVER	Client not present. You forgot to call the setup routine.
SC_BADPROC	Non-existent server procedure called. The RPC protocol got out of sync.
SC_STRTOOLONG	A server procedure returned a string that is longer than specified in the interface definition.
SC_BROKEN	The connection to the client is broken.
SC_BADMSG	A server stub could not decode a request message. The RPC protocol got out of sync.
SC_NODATA	A server stub has attempted to receive more data than arrived in the request message. The RPC protocol got out of sync.
SC_NOMEM	Not enough memory to allocate dynamic parameters.

D User Library Reference

This appendix covers a complete reference to the user library. There are a couple of things to consider when looking up functions in the reference part.

First, if not explicitly mentioned in the description part, the C and Fortran routine behave identical. Note however, that if the C routine returns any value, this value is returned as the last parameter of the Fortran subroutine. There are no Fortran functions.

Second, a routine that returns nothing else, but a return code (`rc`) returns always `SC_OK` (zero) on success.

Third, most routines may also return the error code `SC_SYSCALL` (-1) to indicate that a system call failed. However, this error code is not listed. If it occurs, the global system variable `errno` contains one of the usual UNIX error codes to indicate the type of failure (see the header file `<errno.h>`).

Finally, some of the routine may return the error code `SC_SYSERR` some time, but this should not occur, it is an indication of a bug.

sc_allocarray()

Allocate a n-dimensional C array.

Synopsis

C `int rc = sc_allocarray(char **ap, int num, int *dims, int elsiz)`

Fortran `[not available]`

Parameters

`ap`	generic pointer to array
`num`	number of dimensions
`dims`	length of each dimension
`elsiz`	size of an array element in bytes

Description

The routine `sc_allocarray()` allocates n-dimensional C arrays. A generic pointer to the arrays is passed along with the dimension information and the size of a single array element in bytes. The array may not have more than 16 dimensions.
This routine is not available to Fortran programs, because there is no notion of a pointer in Fortran.

Example

In the following short example program a 4-dimensional array of complex numbers is allocated.

```
sc_complex ****carr;
int dims[4], num, rc;

num = 4;
dims[0] = 5; dims[1] = 3; dims[2] = 7; dims[3] = 2;

rc = sc_allocarray(&carr, num, dims, sizeof(sc_complex));
            :
            :
sc_freearray(carr);
```

Error Codes

Code	Possible Reason
SC_BADPARAM	Attempt to allocate an array with more than 16 dimensions.
SC_NOMEM	Not enough memory to allocate the array.

sc_autoerr() scfautoerr()

Switch automatic error message printing on and off.

Synopsis

C	`void sc_autoerr(int onoff)`
Fortran	`call scfautoerr(onoff)`

Parameters

`onoff`	Boolean switch (0= off, else on)

Description

This routine switches automatic error message printing on and off. When off, error messages can still be printed with the routine `sc_perror()`. The default is on.

Error Codes

Code	Possible Reason
`none`	

sc_calldiscard() scfcalldiscard()

Discard the results of an asynchronous RPC.

Synopsis

C `int rc = sc_calldiscard(int cid)`
Fortran `call scfcalldiscard(cid, rc)`

Parameters

`cid` call identifier

Description

The routine `sc_calldiscard()` discards the results of an asynchronous RPC. The call identifier becomes invalid after this operation.

Note: If the results are not immediately ready to discard, this routine blocks the caller until the results become ready.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call identifier is invalid.
SC_BROKEN	The connection to the server is broken.
SC_NOMEM	Not enough memory.

sc_callgetprocid() scfcallgetprocid()

Get the procedure number for a call.

Synopsis

C `int procid = sc_callgetprocid(int cid)`
Fortran `call scfcallgetprocid(cid, procid)`

Parameters

`cid` call identifier

Description

The routine `sc_callgetprocid()` returns the procedure identifier associated with the call `cid`. Procedure numbers are defined in the common header file produced by the stub generator.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call identifier `cid` is invalid.

sc_callgetsrvid() scfcallgetsrvid()

Get the server identifier for a call.

Synopsis

C `int srvid = sc_callgetsrvid(int cid)`
Fortran `call scfcallgetsrvid(cid, srvid)`

Parameters

`cid` call identifier

Description

The routine `sc_callgetsrvid()` returns the identifier of the server that was called.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call identifier `cid` is invalid.

sc_callisready() scfcallisready()

Check for availability of results of an asynchronous RPC.

Synopsis

C `int ready = sc_callisready(int cid, long timeout)`
Fortran `call scfcallisready(cid, timeout, ready)`

Parameters

`cid`	call identifier
`timeout`	timeout value in milliseconds

Description

The function `sc_callisready()` checks for the availability of results of an asynchronous RPC. A timeout value `timeout` can be specified. The time is in milliseconds. A value of `RDY_BLOCK` (-1L) is equivalent to an infinite timeout and blocks the caller until the results are ready. `RDY_POLL` (OL) polls for readiness. On success, the function returns one (1) if the call is ready and zero (0) if it has been timed out.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call identifier is invalid.
SC_BROKEN	The connection to the server is broken.
SC_NOMEM	Not enough memory.

sc_cgadd() scfcgadd()

Add a call identifier to a call group.

Synopsis

C `int rc = sc_cgadd(int cgid, int cid)`
Fortran `call scfcgadd(cgid, cid, rc)`

Parameters

`cgid`	call group identifier
`cid`	call identifier

Description

The routine `sc_cgadd()` adds a call identifier to a call group.

Error Codes

Code	Possible Reason
SC_BADPARAM	Either the call or the call group identifier is invalid.
SC_NOMEM	Not enough memory.

sc_cgcreate() scfcgcreate()

Create an empty call group.

Synopsis

C `int cgid = sc_cgcreate(void)`
Fortran `scfcgcreate(cgid)`

Parameters

`none`

Description

The routine `sc_cgcreate()` creates an empty call group. On success, it returns a valid call group identifier.

Error Codes

Code	Possible Reason
SC_TABOVFL	Call group table overflow. Too many active call groups.
SC_NOMEM	Not enough memory.

sc_cgdestroy() scfcgdestroy()

Destroy a call group.

Synopsis

C `void sc_cgdestroy(int cgid)`
Fortran `call scfcgdestroy(cgid)`

Parameters

`cgid` call group identifier

Description

The routine `sc_cgdestroy()` destroys a call group. Call identifiers that are still in the group are *not* discarded.

Error Codes

Code	Possible Reason
`none`	

sc_cgdiscardall() scfcgdiscardall()

Discard all calls in a call group.

Synopsis

C `int rc = sc_cgdiscardall(int cgid)`
Fortran `call scfcgdiscardall(cgid, rc)`

Parameters

`cgid` call group identifier

Description

The routine `sc_cgdiscardall()` discards the results of all asynchronous RPCs in the call group. The all call identifiers become invalid and are removed from the call group.

Note: If some results are not immediately ready to discard, this routine blocks the caller until the results become ready.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call group identifier is invalid.
SC_BROKEN	The connection to one of the servers is broken.
SC_NOMEM	Not enough memory.

sc_cgextractready() scfcgextractready()

Extract a ready call from a call group.

Synopsis

C `int cid = sc_cgextractready(int cgid)`
Fortran `scfcgextractready(cgid, cid)`

Parameters

`cgid` call group identifier

Description

The routine `sc_cgextractready()` returns the call identifier of the *first* call to become ready to claim results. It blocks, if no results are immediately ready. The returned call identifier is excluded from the call group.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call group identifier is invalid.
SC_BROKEN	Connection to one of the servers is broken.
SC_NOMEM	Not enough memory.

sc_cggetsize() scfcggetsize()

Get the size of a call group.

Synopsis

C `int size = sc_cggetsize(int cgid)`
Fortran `call scfcggetsize(cgid, size)`

Parameters

`cgid` call group identifier

Description

The routine `sc_cggetsize()` returns the size of a call group, i.e. the number of call identifiers in the group.

Error Codes

Code	Possible Reason
SC_BADPARAM	The call group identifier is invalid.

sc_exception() scfexception()

Raise an exception.

Synopsis

C `int rc = sc_exception(int excptcode)`
Fortran `call scfexception(excptcode, rc)`

Parameters

`excptcode` exception code

Description

The routine `sc_exception()` raises an exception in a server procedure. An exception is defined to be an application level failure of the server procedure. The server procedure is aborted and the server returns to the call dispatcher, ready to accept new RPC requests.

Exceptions are useful to indicate the client a failure of a server procedure without the need to send back useless return values. Only the exception code is sent back and the client gets back the error code `SC_RPCFAILED` from its call/claim stub. The system variable `sc_rpcret` is set to the exception code. Exception codes range from 1 to 31.

Note: This routine returns only in case of an error.

Warning: Raising an exception from within a signal handler can not be guaranteed to work on all systems.

Error Codes

Code	Possible Reason
SC_BADPARAM	The exception code is not in the range of 1 to 31 or the routine was called by the master.

sc_freearray()

Free a n-dimensional C array.

Synopsis

C `int rc = sc_freearray(char *ap)`
Fortran `[not available]`

Parameters

`ap` generic array pointer

Description

The routine `sc_freearray()` frees array memory that was previously allocated with `sc_allocarray()`.
The routine is available to Fortran program.

Example

See `sc_allocarray()`.

Error Codes

Code	Possible Reason
none	

sc_initmaster() scfinitmaster()

Initialize the master.

Synopsis

C `int rc = sc_initmaster(int *argc, char **argv)`
Fortran `call scfinitmaster(rc)`

Parameters

argc	reference to number of command line parameters
argv	command line parameters

Description

The top-level client process is called the master. This routine must be called to initialize the master. This routine reads parameters from the command line. In particular, it checks for the `-m` option which specifies the monitor address. The synopsis is different in C and Fortran, because command line parameters are accessed differently from the two languages.

In C, the arguments are a reference to the number of arguments and the arguments themselves. The routine `sc_initmaster()` checks for the `-m` option in the command line and removes it from the argument list.

The Fortran routine, however, needs no parameters (except for the almost omnipresent return code) and does NOT filter out the `-m` option from the command line.

Error Codes

Code	Possible Reason
SC_BADOPTION	The monitor option `-m` was not specified correctly.
SC_NOHOSTFILE	The host file `.sc_hosts` in your home directory could not be opened.
SC_BADHOSTFILE	There is a format error in the host file.
SC_NOHOST	A host listed in the host file is not known to the system.
SC_TABOVFL	Host table overflow. Too many hosts listed in the host file.

sc_installcontext() scfinstallcontext()

Synopsis

C `int rc = sc_installcontext(int num, char *cvp, ...)`
Fortran `call scfinstallcontext(rc, num, cvp, ...)`

Parameters

`num`	number of context variables to install
`cvp`	reference to context variable (`num` parameters)

Description

The routine `sc_installcontext()` stores references to context variables away for later reference by the server stub. The first argument is the number of context variables to install and the following parameters are references to context variables. The number `num` must match the number of context variables in the interface definition. The references must appear in the order of declaration in the interface definition.

Note: The Fortran version `scfinstallcontext()` returns the return code `rc` as the first parameter.

Example

Consider the following interface definition:

```
INTERFACE ContextExample;

CONTEXT
    b: BLOCK
           i: INTEGER;
           a: ARRAY [10] OF LONGREAL;
       END;

PROCEDURE UpdateContext(IN b: ^CONTEXT);

END
```

Example (cont.)

A corresponding server program in Fortran might look like:

```
program example
common /b/i, a
integer i, rc
double precision a(10)

call setupfContextExample(rc)
call scfinstallcontext(rc, 1, b)

call runfContextExample(rc)
end
```

After the setup the common block `b` is installed as a context variable. Then, the server is started by calling the run stub routine. The client may update the context block `b` by calling the client stub call_UpdateContext(). Note that no server procedure `UpdateContext()` will be called, since `UpdateContext` is a pure context transfer procedure, which means it has exclusively context parameters.

Error Codes

Code	Possible Reason
SC_BADPARAM	Negative number passed for `num` or calling program not a server.
SC_NOMEM	Not enough memory.

sc_perror() scfperror()

Print error message.

Synopsis

C `void sc_perror(char *msg)`
Fortran `call scfperror(msg)`

Parameters

`msg` user message to be printed

Description

This routine prints the message `msg` and appends the error message associated with the error variable `sc_errno`.

Error Codes

Code	Possible Reason
`none`	

sc_setbufsize() scfsetbufsize()

Change the size of the send and receive buffers.

Synopsis

C `int rc = sc_setbufsize(int size, int which)`
Fortran `call scfsetbufsize(size, which, rc)`

Parameters

`size`	new size of the buffer(s)
`which`	which buffer(s); send (1), recv (0) or both (2)

Description

The routine `sc_setbufsize()` resizes the message buffers. The parameter `which` determines which buffers should be resized. A value of 0 means the receive buffer only, 1 means the send buffer only and 2 means both, send and receive buffers. The size of these buffers has a direct influence on how many times the system calls `send(2)` and/or `recv(2)` are called to transmit a large message. Since these system calls are quite time-consuming, increasing the size of these buffers increases data transfer performance for large messages.

Error Codes

Code	Possible Reason
SC_BADPARAM	`which` is different from 0, 1, 2.
SC_NOMEM	Not enough memory to allocate the buffer(s).

sc_softerr() scfsofterr()

Switch soft errors on and off.

Synopsis

C `void sc_softerr(int onoff)`
Fortran `call scfsofterr(onoff)`

Parameters

`onoff` Boolean switch (0 = off, else on)

Description

This routine switches between soft and hard errors. When soft errors are on, library routines return an error code, whereas with hard errors the program is aborted on the occurrence of an error. In this case, the exit code of the program is the absolute value of the error code. Hard errors are useful for debugging purposes. The default value is soft errors on.

Error Codes

Code	Possible Reason
none	

sc_srvgethost() scfsrvgethost()

Get the host name for a server.

Synopsis

C `int rc = sc_srvgethost(int srvid, char *host)`
Fortran `call scfsrvgethost(srvid, host, rc)`

Parameters

`srvid`	server identifier
`host`	name of the server host

Description

The routine `sc_srvgethost()` returns the name of the host the server with id `srvid` is running on.

Error Codes

Code	Possible Reason
SC_BADPARAM	`Srvid` is not a valid server identifier.

sc_srvsendsig() scfsrvsendsig()

Send a UNIX signal to a server.

Synopsis

C `int rc = sc_srvsendsig(int srvid, int signo)`
Fortran `call scfsrvsendsig(srvid, signo, rc)`

Parameters

`srvid`	server identifier
`signo`	UNIX signal number

Description

The routine `sc_srvsendsig()` sends a UNIX signal number `signo` to the server `srvid`. See header file `signal.h` for the valid signal names and numbers. If there is no signal handler installed in the server program, the default action will be taken, which terminates the process in most cases (see also kill(2) and signal(3)). Signals are delivered to servers in cooperation with the super-server.

Error Codes

Code	Possible Reason
SC_BADPARAM	Invalid signal number passed. See header file `<signal.h>`
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.

Error Codes (cont.)

Code	Possible Reason
SC_NOHOST	Host where server `srvid` is running is not known to the system.
SC_NOSERVER	`Srvid` does not denote a valid server.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.

sc_srvstart() scfsrvstart()

Start an application server.

Synopsis

C
```
int sid = sc_srvstart(char *host, char *svcname,
                      int niceval)
```

Fortran
```
call scfsrvstart(host, svcname, niceval, sid)
```

Parameters

host	name of server machine
svcname	name of the service
niceval	nice value for lowering server priority

Description

The routine `sc_srvstart()` starts a server exporting the service `svcname` on the machine `host`. A nice value `niceval` greater than zero may be specified in order to lower the server priority (see nice(1)). On success, the routine returns a server identifier.

The server is started in cooperation with the super-server. The super-server gets the name of the executable from the services file `.sc_services` in the user's home directory. There must be an entry for the service `svcname` in that file on the server machine `host`. Moreover this file must not be readable by any other than the user.

Error Codes

Code	Possible Reason
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.

Error Codes (cont.)

Code	Possible Reason
SC_REQFAILED	Super-server failed to process request. Probably some failed system call. Check the super-server log file for the exact reason.
SC_NOEXEC	there is no services file `.sc_services` on the server host; there is no entry for the service `svcname`; the listed executable could not be found or is not executable; the services file is readable by others than the user - see super-server's log file for the exact reason.
SC_NOHOST	Host is not known to the system.
SC_NOSERVICE	The service is not known to the RTS. Probably it has not been attached correctly.
SC_STUBVERS	Stub version mismatch. The client and server stubs do not stem from the same version of the interface definition file.
SC_RTSVERS	RTS version mismatch. The client and server programs have been linked to different versions of the library.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.
SC_NOMEM	Not enough memory available to perform the operation.
SC_TABOVFL	Server table is full. There are too many servers running.

sc_srvstartconfig() scfsrvstartconfig()

Start server configuration.

Synopsis

C
```
int rc = sc_srvstartconfig(char *filenam, int *niceval,
                           int *numinst, int *srvids)
```

Fortran
```
call scfsrvstartconfig(filenam, niceval,
                       numinst, srvids, rc)
```

Parameters

`filenam`	name of the server configuration file
`niceval`	nice value
`numinst`	length of srvids (in), number of started servers (out)
`srvids`	server identifiers

Description

The routine `sc_srvstartconfig()` starts a configuration of servers. The configuration is listed in a file. The format of the file is as follows:

```
<host-name>   <#inst>   <service-name>
```

where `<host-name>` is the name of the server host, `<#inst>` is the number of instances to be started and `<service-name>` is the name of the service. Empty lines and lines starting with the hash mark (#) are ignored.

A nice value `niceval` greater than zero may be specified in order to lower the server priorities (see nice(1)). The routine returns the number of started servers in `numinst` and the server identifiers in `srvids`. On input, `numinst` must contain the length of the `srvids` array.

The servers are started in cooperation with the super-server. The super-server gets the name of the executable from the services file `.sc_services` in the user's home directory. There must be an entry for the service `svcname` in that file on the server machine `host`. Moreover this file must not be readable by any other than the user.

Example

Here is an example for a configuration file named `MyApp.conf`:

```
# configuration file for application 'MyApp'
monty       3   FLYING
python      4   CIRCUS
racehorse   7   RUNFAST
```

Example (cont.)

The following code fragment starts the configuration:

```
int sids[16], ninst, rc;

ninst = 16;  /* do not forget this */
rc = sc_srvstartconfig("MyApp.conf", NICEVAL, &ninst, sids);
```

Error Codes

Code	Possible Reason
SC_BADPARAM	There are more servers listed in the configuration file than `numinst` on input indicates for the length of the `srvids` vector; there is a line with `<#inst>` greater than 16 servers in the configuration file.
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.
SC_REQFAILED	Super-server failed to process request. Probably some failed system call. Check the super-server log file for the exact reason.
SC_NOEXEC	there is no services file `.sc_services` on the server host; there is no entry for the service `svcname`; the listed executable could not be found or is not executable; the services file is readable by others than the user - see super-server's log file for the exact reason.
SC_NOHOST	Host is not known to the system.
SC_NOSERVICE	The service is not known to the RTS. Probably it has not been attached correctly.

Error Codes (cont.)

Code	Possible Reason
SC_STUBVERS	Stub version mismatch. The client and server stubs do not stem from the same version of the interface definition file.
SC_RTSVERS	RTS version mismatch. The client and server programs have been linked to different versions of the library.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.
SC_NOMEM	Not enough memory available to perform the operation.
SC_TABOVFL	Server table is full. There are too many servers running.

sc_srvstartmulti() scfsrvstartmulti()

Start multiple application servers.

Synopsis

C
```
int rc = sc_srvstartmulti(char *host, char *svcname,
                          int niceval,
                          int *numinst, int *srvids)
```

Fortran
```
call scfsrvstartmulti(host, svcname, niceval,
                      numinst, srvids, rc)
```

Parameters

`host`	name of server host
`svcname`	name of service
`niceval`	nice value
`numinst`	number of instances
`srvids`	server identifiers

Description

The routine `sc_srvstartmulti()` starts `numinst` servers exporting the service `svcname` on the machine `host`. A nice value `niceval` greater than zero may be specified in order to lower the server priorities (see nice(1)). On success, the server identifiers are returned in `srvids`.

The servers are started in cooperation with the super-server. The super-server gets the name of the executable from the services file `.sc_services` in the user's home directory. There must be an entry for the service `svcname` in that file on the server machine `host`. Moreover this file must not be readable by any other than the user.

Error Codes

Code	Possible Reason
SC_BADPARAM	`numinst` is greater than 16.
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.

Error Codes (cont.)

Code	Possible Reason
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.
SC_REQFAILED	Super-server failed to process request. Probably some failed system call. Check the super-server log file for the exact reason.
SC_NOEXEC	there is no services file `.sc_services` on the server host; there is no entry for the service `svcname`; the listed executable could not be found or is not executable; the services file is readable by others than the user - see super-server's log file for the exact reason.
SC_NOHOST	Host is not known to the system.
SC_NOSERVICE	The service is not known to the RTS. Probably it has not been attached correctly.
SC_STUBVERS	Stub version mismatch. The client and server stubs do not stem from the same version of the interface definition file.
SC_RTSVERS	RTS version mismatch. The client and server programs have been linked to different versions of the library.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.
SC_NOMEM	Not enough memory available to perform the operation.
SC_TABOVFL	Server table is full. There are too many servers running.

sc_srvterm() scfsrvterm()

Terminate a server.

Synopsis

C `int rc = sc_srvterm(int srvid)`
Fortran `call scfsrvterm(srvid, rc)`

Parameters

`srvid` server identifier

Description

Terminates the server `srvid`. Before the server exits it cleans up and thereby detaches all its services which causes any subordinate servers to be terminated. This action is performed in cooperation with the super-server.

Error Codes

Code	Possible Reason
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.
SC_NOHOST	Host where server `srvid` is running is not known to the system.
SC_NOSERVER	`Srvid` does not denote a valid server.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.

sc_svcattach() scfsvcatatch()

Attach a service.

Synopsis

C
```
int rc = sc_svcattach(char *svcname,
                      void (*svcinit)(long *)))
```

Fortran
```
call scfsvcattach(svcname, svcinit, rc)
```

Parameters

`svcname`	a string identifier for the service
`svcinit`	service initialization routine exported by the client stub

Description

The routine `sc_svcattach()` attaches a service to a client program. It assigns the service a string identifier `svcname` which can be used to reference the service later. This identifier is also referenced in the service file `.sc_services` which maps service names to executable path names. The routine `svcinit` is an initialization routine exported by the client stub. Its name is composed of the prefix `init_` and the name of the interface definition, i.e. the identifier appearing after the keyword `INTERFACE` in the interface definition file.

Example

The service defined by the interface definition

```
INTERFACE Bla;
...
END
```

could be initialized with the following call

```
rc = sc_svcattach("BLASVC", init_Bla);
```

The service is now named `BLASVC`

Error Codes

Code	Possible Reason
`SC_NOMEM`	The process out of memory.
`SC_TABOVFL`	The client has too many services attached to it.

sc_svcdetach() scfsvcdetach()

Detach a service.

Synopsis

C `int rc = sc_svcdetach(char *svcname)`
Fortran `call scfsvcdetach(svcname)`

Parameters

`svcname` name of the service to detach

Description

The routine `sc_svcdetach` detaches the specified service from the client. After this operation the service is not known any longer. Servers that are still running are terminated. A warning message is printed, if one of the servers is still busy processing a remote call (see also sc_srvterm()).

Error Codes

Code	Possible Reason
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.
SC_NOSERVICE	The string identifier `svcname` is not referencing a service that has been successfully attached with `sc_svcattach()`
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.

sc_svcdetachall() scfsvcdetachall()

Detach all services.

Synopsis

C `int rc = sc_svcdetachall(void)`
Fortran `call scfsvcdetachall(rc)`

Parameters

`none`

Description

The routine `sc_svcdetachall()` detaches all services from a client that have been previously attached with `sc_svcattach()`. After this operation no service is known any longer. Any servers that are still running are terminated. A warning message is printed, if one of the servers is still busy processing a remote call (see also sc_srvterm()).

Error Codes

Code	Possible Reason
SC_NODAEMON	There is no super-server running on the server host.
SC_NOMONITOR	Can't connect to monitor. You probably forgot to pass the monitor address to the master or you passed a wrong address.
SC_AUTHFAILED	Authentication failure. Maybe, there is a wrong login name entry in the host configuration file; you entered a wrong password; you do not have an account on `host`.
SC_REQREJECTED	You connected to the super-server of another user. Check the port number in the host configuration file. Could also be a system error in the super-server. Check the super-server log file for the exact reason.
SC_BROKEN	The connection to the super-server is broken. Maybe the super-server or its host have crashed.
SC_TIMEOUT	There is a problem with the connection to the super-server.

scferrno()

Get `sc_errno` variable from Fortran program.

Synopsis

C `[not needed]`
Fortran `call scferrno(errno)`

Parameters

`errno` value of `sc_errno`

Description

This routine returns the value of the variable `sc_errno`, the error code of the last operation that failed, to the calling Fortran program.

Error Codes

Code	Possible Reason
`none`	

scfrpcret()

Get `sc_rpcret` variable from a Fortran program.

Synopsis

C `[not needed]`
Fortran `call scfrpcret(rpcret)`

Parameters

`rpcret` value of `sc_rpcret`

Description

This routine returns the value of the system variable `sc_rpcret`, the return value of the last RPC, to the calling Fortran program.

Error Codes

Code	Possible Reason
`none`	

E Runtime System Error Codes and Messages

Mnemonic	Code	Message
SC_OK	0	No Error
SC_SYSCALL	-1	System Call Failed
SC_BADPARAM	-2	Bad Parameter
SC_BADOPTION	-3	Bad Command Line Option
SC_NOHOSTFILE	-4	Host File Not Found
SC_BADHOSTFILE	-5	Bad Host File Format
SC_NOCONFIGFILE	-6	Cannot Open Configuration File
SC_BADCONFIGFILE	-7	Bad Configuration File Format
SC_NODAEMON	-8	No Daemon Present
SC_NOMONITOR	-9	No Monitor Available
SC_AUTHFAILED	-10	Authentication Failed
SC_REQREJECTED	-11	Request Rejected by Super-Server
SC_REQFAILED	-12	Super-Server Failed to Process Request
SC_NOHOST	-13	No such Host
SC_NOEXEC	-14	No such Executable
SC_NOSERVICE	-15	No such Service
SC_NOSERVER	-16	No such Server
SC_STUBVERS	-17	Stub Version Mismatch
SC_RTSVERS	-18	RTS Version Mismatch
SC_BADPROC	-19	Bad Procedure Number
SC_RPCFAILED	-20	RPC Failed
SC_BADCLAIM	-21	Wrong Claim Stub
SC_STRTOOLONG	-22	String too Long
SC_BADVIEW	-23	Illegal View
SC_BROKEN	-24	Connection Broken
SC_TIMEOUT	-25	Timed Out
SC_BADMSG	-26	Can't Decode Message
SC_NODATA	-27	Read Past End of Message
SC_NOMEM	-28	Out of Memory
SC_TABOVFL	-29	Internal Table Overflow
SC_SYSERR	-30	System Error

F File Formats

Host File

The host file is named `.sc_hosts` and must reside in your home directory. It contains a list of hosts and the super-server ports and must be login names on those hosts. It is read at runtime system initialization time. The format of a line in this file is:

```
<host-name>  <super-server-port>  [<login-name>]
```

Lines starting with a blank or with the hash mark (`#`) are treated as comment lines. If the login name is omitted, it is assumed to be the same as on the local host. There are two special port numbers: a hash mark (`#`) denotes the super-server default port in multi-user mode and an asterisk (`*`) the default port in user mode.

Services File

The services file is named `.sc_services` and must reside in your home directory. It maps service names to executables. It is read by the super-server to find the executable for a service. Note, that for security reasons this file must not be readable or writable by any other user than you. The format of a line in this file is:

```
<service-name>  <executable-path-name>
```

Lines starting with a blank or with the hash mark (`#`) are treated as comment lines. Note that relative paths start from the home directory. A percent sign (`%`) in the path is expanded to the architecture tag of the local machine.

Server Configuration File

Server configuration files are passed to the routine `sc_srvstartconfig()`. They list how many instances of which services are to be started on which machines. Their line format is:

```
<host-name>  <#instances>  <service-name>
```

Lines starting with a blank or with the hash mark (`#`) are treated as comment lines.